Published in: Proceeding of 6th International Symposium on Representations and Methodology of Future Computing Technology (RM2003), Trier, Germany, 10-11 March 2003, pp 63-68

BDDs, Horn Clauses and Resolution

Mohammad GhasemZadeh
FB IV-Informatik, University of Trier
D-54286 Trier, Germany
GhasemZadeh@TI.Uni-Trier.DE

Christoph Meinel
FB IV-Informatik, University of Trier
D-54286 Trier, Germany
Meinel@Uni-Trier.DE

Abstract

In this research we present the utilization of BDDs in representing propositional logic programs and implementing the refutation by **resolution** deduction method.

A logic program is a collection of axioms from which a goal clause can be proven. Axioms are written in a standard form known as **Horn clauses**. In logic programming, we try to find a collection of axioms and inference steps that imply the goal. The standard and usual method for this kind of inference is 'refutation by **resolution'.**

Binary Decision Diagrams, shortly called BDDs, are data structures proposed for representing switching functions. BDDs have been found more practical and more efficient in time and space than other switching function representation methods.

One may consider a Horn clause as a switching function and represent it as a BDD. In the same way, all the clauses of a propositional logic program and the goal clause can be represented by means of a multi-rooted BDD. Because of the characteristics of BDDs we can make the inference in some other methods in addition to the standard method. In this paper, this kind of representation as well as proving the goal in formal linear resolution and Non-linear resolution are investigated.

Keywords: Logic programming, Horn clauses, Binary Decision Diagrams-BDDs, Normal Forms, Resolution .

1 Introduction

In logic programming, the programmer specifies what is known and what is the question to be solved. Logic programming is also known as declarative programming. This kind of programming is widely used for Artificial Intelligence and knowledge based systems (non-numeric Processing).

PROLOG (PROgramming in LOGic) is the popular Logic Programming Language. PROLOG and most of the other declarative programming tools use resolution with backward chaining in order to prove the goal. Most backtracking algorithms need exponential run time and memory space.

BDDs are data structures proposed for representing switching functions [3,4]. These data structures have been extensively studied and have provided new techniques powerful for verification and synthesis of switching functions. In many cases where other methods give exponential solutions, the use of BDDs resulted in a polynomial solution or at least a high decrease in run time and needed memory space.

A Logic program consists of a list of Horn clauses. One can look at each clause as a switching function and represent it as a BDD. So the idea and question emerges: Is it possible to use BDDs also in logic programming to obtain better results ?

In this paper we introduce the presentation of a logic program and the goal by means of a multi rooted BDD and we present two methods for proving the goal.

2 Programming with Propositional Logic

2.1 Definitions

A **propositional literal** is a propositional variable or the negation of a propositional variable. A **positive** propositional literal is a propositional variable and a **negative** propositional literal is the negation of a propositional variable.

A **propositional clause** is a disjunction of pro- positional literals; it is a propositional formula where a number of propositional literals are connected by the $\vee$ operator. Any propositional formula can be written in terms of a conjunction of clauses, a number of clauses connected together by the $\wedge$ operator. This form is called **Conjunction Normal Form (CNF)**. An example of a propositional clause is: $p \vee \neg q \vee r \vee \neg s$, where p, q , r and s are propositional variables.

We now consider a certain type of clause, known as Horn clause, this class of clauses is particularly important for logic programming. A **propositional Horn clause** is a propositional clause with at most

one positive literal. A Horn clause is thus in one of the following three forms:

1) q
2) $\neg p_1 \vee \ldots \vee \neg p_n \vee q$
3) $\neg p_1 \vee \ldots \vee \neg p_n$

Where $p_1, \ldots, p_n, q$ are propositional variables.

Horn clauses of the forms outlined in (1) and (2) with one positive literal are known as **program clauses** Horn clauses of the form outlined in (1) are sometimes called **unit clauses**. Notice that the Horn clause given in (2) can be rewritten as :

$$\neg(p_1 \wedge \ldots \wedge p_n) \vee q$$

and this in turn can be rewritten as:

$$(p_1 \wedge \ldots \wedge p_n) \rightarrow q$$

Horn clauses of the type outlined in (3) are known as **goal clauses**. The Horn clause given in (3) can be rewritten as : $\neg(p_1 \wedge \ldots \wedge p_n)$

2.2 Propositional Resolution

Suppose we have $C_1 \vee p$ and $C_2 \vee \neg p$, where C_1 and C_2 are clauses and p is a propositional variable. The rule of **resolution** allows us to obtain an expression for $C_1 \vee C_2$ which does not involve the variable p. We can define resolution as the following inference rule :

$$C_1 \vee p \ , \ C_2 \vee \neg p \ \vdash_{Res} \ C_1 \vee C_2$$

where the symbol $\vdash_{Res}$ denotes deduction using only the resolution rule. In other words, we use resolution to derive new statements from existing statements. For instance :

$$\neg A \vee \neg B \vee C, \ \neg C \vee D \vdash_{Res} \neg A \vee \neg B \vee D$$

which is the same as :

$$A \wedge B \rightarrow C \ , \ C \rightarrow D \ \Rightarrow \ A \wedge B \rightarrow D$$

"If we know that A and B imply C, and C implies D, then we can deduce that A and B imply D."

2.3 Refutation and Deduction

Resolution is the rule of inference that we use in logic programming to perform deductions. A logic program consists of a set of program clauses, which can be considered as a set of hypotheses. The resolution rule can be applied to the hypotheses to deduce consequents. There is a particular method used in logic programming to perform these deductions. This method is known as **refutation** and also known as **contradiction**.

Suppose we have a logic program. The program can be presented with **queries**. These are statements that are conjunctions of propositional variables. The question being asked of the program is: 'Does the query follow from the program ?' this is equivalent to: 'Given a logic program P (which is a set of program clauses) and a query Q, can we establish the deduction : $P \vdash_{Res} Q$?

Now although the only rule to be used in the deduction is resolution, the method of establishing the deduction is by refutation (contradiction). We will add $\neg Q$ as a hypothesis and use resolution to establish a contradiction in the form of the **empty clause**.

Using propositional resolution, the only way to obtain the empty clause is by applying the rule to a propositional variable p and $\neg p$, in which case we have a contradiction. Our contradiction is really established in the step before the empty clause is reached. The empty clause is denoted by .

In logic programming we are interested in establishing deductions of the form: $P \vdash_{Res} Q$ and the way we do this is by first establishing a deduction of the form: $P, \neg Q$. This method is called **refutation by resolution**.

A query is a conjunction of propositional variable, it will be in the form: $p_1 \wedge \ldots \wedge p_n$, its negation: $\neg p_1 \vee \ldots \vee \neg p_n$, would be in the form of a goal clause (Horn clause type 3). In our definitions a negative literal cannot be a query. This is far too restrictive and some methods of handling negative queries will be needed later on. **Negation in logic programming** is surprisingly complicated.

2.4 Formal Resolution Deduction

Suppose we have a set of propositional program clauses and a query. The query, upon negation, becomes a goal clause. Resolution is then applied to the hypotheses in an attempt to derive the empty clause. Which clauses are to be combined in each step? Prolog tries to solve the left most variable of the goal first, also at a certain point in the deduction, it always chooses the clause which is listed first; If the wrong choice is taken then it ignores what was done from the moment the choice was made and considers the next option. This procedure is known as **resolution with backtracking**.

Suppose we have the following program :

(1) $p \wedge r \wedge s \rightarrow q$	(1) $\neg p \vee \neg r \vee \neg s \vee q$
(2) $p \wedge w \rightarrow q$	(2) $\neg p \vee \neg w \vee q$
(3) $p \wedge t \rightarrow n$	(3) $\neg p \vee \neg t \vee n$
(4) $w \rightarrow n$	(4) $\neg w \vee n$
(5) $s \rightarrow w$	(5) $\neg s \vee w$
(6) $t \rightarrow r$	(6) $\neg t \vee r$
(7) p	(7) p
(8) t	(8) t
(9) s	(9) s

also, suppose it is presented with the Query: $q \wedge n \wedge w$ which means that $\neg q \vee \neg n \vee \neg w$ should be added as an extra hypothesis. The program could be re-

written in terms of ¬ and ∨. Then the resolution deduction carried out as follows :

(10)	¬q∨¬n∨¬w	Negation of the Quary
(11)	¬p∨¬r∨¬s∨¬n∨¬w	Res. 10,1
(12)	¬r∨¬s∨¬n∨¬w	Res. 11,7
(13)	¬t∨¬s∨¬n∨¬w	Res. 12,6
(14)	¬s∨¬n∨¬w	Res. 13,6
(15)	¬n∨¬w	Res. 14,9
(16)	¬w∨¬w	Res. 15,4
(17)	¬w	
(18)	¬s	Res. 17,5
(19)		Res. 18,9

The empty clause has been deduced, so the hypotheses combined with negation of the query are unsatisfiable, so the query: q∧n∧w **succeeds**.
Figure 1 displays the **resolution tree.**

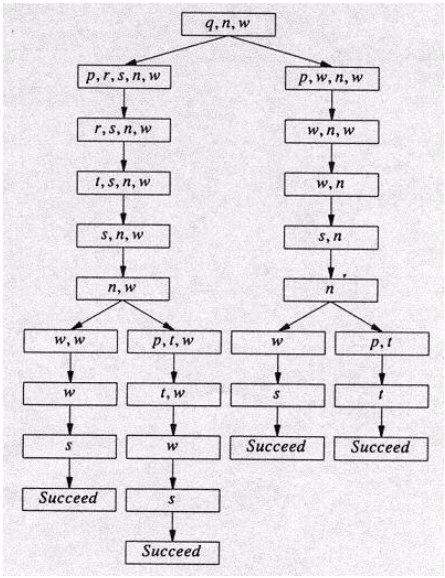

Figure 1

3 Binary Decision Diagrams

3.1 Definitions

A **Binary Decision Diagram** is a rooted, directed, acyclic graph with the following properties:

- One or more roots;
  Each root represents a boolean function
- Two distinct terminal nodes labeled with the Boolean constants 0 and 1
- Each non-terminal node is labeled with a boolean variable X_i and has two outgoing edges labeled with 0 and 1

Figure 2, displays a BDD representing of the boolean function: **F=(x_1 + x_2). x_3**

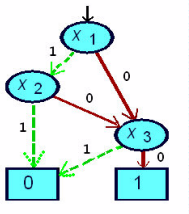

Figure 2

You can easily see how does an OBDD represent a switching function.

BDDs are proposed as data structures for representing switching functions by Lee as early as 1959 [1], and later by Akers [2]. In 1986, Bryant [3] showed that tasks for manipulating switching functions can be performed very efficiently if some variable ordering restrictions on the BDD structure are satisfied.

For this reason, Bryant proposed to employ Ordered Binary Decision Diagrams (OBDDs). In an OBDD, on each path, each variable is evaluated at most once and in the specified variable order.

OBDDs have found more practical applications than other representations of switching functions, mainly for two reasons (see e.g. [4]). First, by applying reduction algorithms, OBDDs can be transformed into a canonical form, which **uniquely** characterize a given function. Second, in terms of their representation, Boolean operations on switching functions can be performed quite efficiently in time and space. For example, an AND composition of two switching functions can be performed in time that is linear in the product of the sizes of their OBDD representations.

3.2 The ITE(, ,) Algorithm

The usual way of generating new BDDs is to combine existing BDDs with connectives like AND, OR, EX-OR. If we want to make an OBDD for a given boolean function, first we make OBDDs for each variable of the Boolean function, and then we parse the Boolean function and combine the existing OBDDs to make OBDDs for the needed subfunctions and finally the OBDD representing the whole given Boolean function.

ITE(, ,) is the most important algorithm for combining and constructing OBDDs. This procedure is a combination of **dynamic programming** and depth-first traversal. It receives OBDDs for two Boolean functions F and G, builds the OBDD for F<op>G. All two-argument operators can be expressed in term of ITE For example AND(F,G) is ITE(F, G, 0) ; and Not(G) is ITE(G, 0, 1).

This procedure maintains a table called **Computed Table** to avoid computing a subfunction many times, also it maintains another table called **Unique Table** to avoid subgraphs representing the same subfunctions. The benefit of this technique is the important result that ITE(, ,) is a **polynomial** algorithm. Notice that the number of times we call ITE(, ,) can be exponential and this would make the whole computation exponential.

However, OBDDs share a fatal property with all kinds of representations of switching functions: the representation of almost all functions need exponential space.

There is another problem that occurs when representing switching functions by means of OBDDs: the size of an OBDD (number of nodes), depends on the order of the input variables. Figure 3, shows the effect of variable ordering for a switching function. Both OBDDs represent the same Boolean function : $F = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

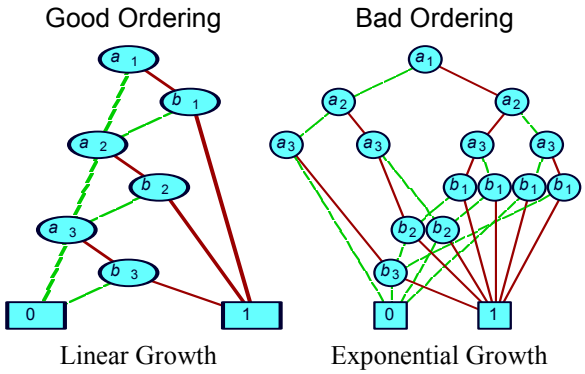

Figure 3

Different functions have different ordering sensitivities. Some functions have a high and others have a low variable order sensitivity.

The practicability of OBDDs strongly depends on the existence of suitable algorithms and tools for minimizing the graphs in the relevant applications. Nowadays there exit many improvements, optimization algorithms and additions to the basic OBDD model.

By experience we know: Many tasks have reasonable BDD representations; algorithms remain practical for up to 100,000 node OBDDs; and most proposed heuristic ordering methods are generally satisfactory.

Because of the practical applicability of this data structure, the investigation and development of new optimization techniques for OBDDs is still a rewarding research topic. For a detailed overview on OBDDs and their applications see[4].

4 Logic Programming Using OBDDs

4.1 Representing a Logic Program as an OBDD

Now we know that a propositional logic program is a finite set of Horn clauses and a goal clause of which we are interested to see whether it can be deduced or not. We can easily look at each Horn clause as a Boolean function and represent it as an OBDD.

For example: ¬p∨¬r∨¬s∨q which is a Horn clause type 2, can be represented in the form of the OBDD displayed in Figure 4. Here we have considered the variable order: $\pi(P) < \pi(Q) < \pi(R) < \pi(S)$; but due to the form of Horn clauses, the shape and size of our OBDD would be variable order independent.

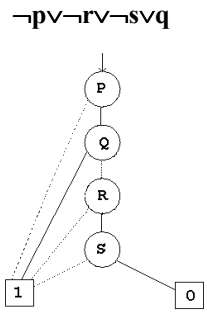

Figure 4

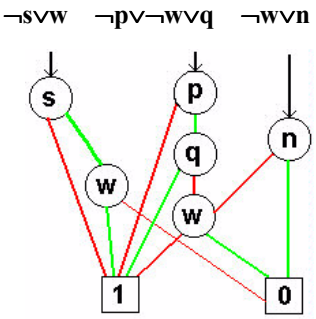

Figure 5

An OBDD may have several roots to present a number of Boolean functions. Therefore we can represent other clauses of our propositional logic program and the goal clause in the same OBDD to have a multi rooted OBDD that represents all the program clauses and the goal clause. Figure 5, displays an OBDD representing three program clauses.

Here we can see the node and the subfunction sharing property of our data structure. This property is very important, specially when we have a big number of clauses. This time, the variable order and the order on which the clauses are added to the OBDD is important and may have a considerable effect.

4.2 Deduction in an OBDD-Based Logic Program

The main purpose of a logic programming system is deducing the goal clause from the set of program clauses.

The standard method for this purpose is SLD-resolution with backward chaining. The SLD stands for Linear resolution with Selection function for Definite clauses. We may apply this method for our OBDD-Based logic programs. (But with some modifications).

OBDD is a non-linear data structure, therefore we can also try non-linear selection functions. This is the second method of deduction we are considering.

From logic we know that if the set of program clauses with the negation of the goal clause are inconsistent, then the conjunction of them will be 0. This observation leads us to the third method of OBDD-based deduction.

4.2.1 Resolving a Clause with Query (Goal)

Here we will see how resolution could be in an OBDD-based logic program.

Suppose **G=¬w∨ ¬q ∨ ¬s** be the negation of the goal we are going to solve. We will have G in our OBDD as a root. The root node of this OBDD will be the variable of G with the lowest variable order. (the variable order of the OBDD). If we had considered the OBDD variable order the same as the alphabet order, then it would be better to look at our goal clause as:

G = ¬q ∨ ¬s ∨ ¬w

This means that **q** will be at the root node of G. Now we consider **¬q** to be resolved with one or more program clauses.

Now, suppose that we have the program clause:

C = ¬p ∨ q ∨ ¬s

We can resolve G and C according to q and ¬q , to obtain another clause:

G = ¬q ∨ ¬s ∨ ¬w ,

C = ¬p ∨ q ∨ ¬s $\vdash_{Res}$ **R = ¬p ∨ ¬s ∨ ¬w**

For this purpose, we must remove **¬q** from **G** to obtain **G'=¬s ∨ ¬w**, remove **q** from **C** and obtain **C'=¬p∨ ¬s** then compute: **R=G'∨C'**

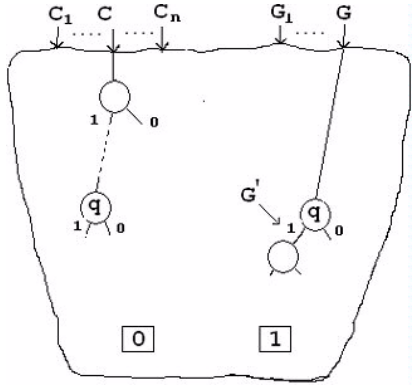

Figure 6

But we are not allowed to remove any node from the OBDD, because it affects other functions (clauses) that are represented in the multi rooted OBDD**.**

According to OBDD structure, since q is at the root node, G' will be the One-successor of G. Therefore we may have G' without any computations. Unfortunately we don't have the same condition for C to obtain C'.

The resolving variable q could easily be somewhere at the middle of the OBDD representing clause C. At this time it seems that we must build C' from C. Figure 6 illustrates this concept.

4.2.2 Standard SLD-Resolation Deduction

As we have seen earlier, this method is the same as working in a depth first manner from left to right in the resolution tree.

A query can be a single or a conjunctions of several variables. There will often be more than one variable which matches the head of one of the program clauses. In this instance, from the point of view of automating the deduction process, it is reasonable to introduce a selection function which picks out the variable to consider from the goal clause. If this succeeds, then the function picks out the next variable and so on. The use of this selection function with resolution is called SLD-resolution.

The selection function used in most Prolog implementations is to pick the left most subgoal (the left most literal of the goal clause.) Since we are representing a logic program as an OBDD with a specific variable order, as we mentioned above it's better to pick the subgoal variable according to our OBDD **variable order**.

In order to implement this method, first we consider a variable order, then we input each program clause and the goal clause, and make OBDDs for each of them. The OBDDs are not distinct OBDDs. We will always have only one multi rooted OBDD. This decision, because of the node and subfunction sharing property in OBDDs, is very important

After reading the clauses and making the multi rooted OBDD, we will have a list of pointers that each of them points to the root of a program clause, also a pointer that points to the root of the primary goal clause. Here we may use a stack to implement our resolution with backward chaining method. Figure 7, displays an abstract interpreter for this purpose.

Input: A logic program **P** and a goal **G** presented as a multi rooted **OBDD. (Program clauses pointer list and a pointer for the goal)**

Output: Yes, if a proof of G from P was found, otherwise No.

Algorithm

Initialise the resolvent stack to be **G.** (the input goal)
/* considering the variable order: A_1,..., A_n */
while the resolvent stack is not empty
{
Pop a goal clause **G = ¬A_1∨ . . .∨ ¬A_n**
if G is empty **exit**
for all program clauses **C:¬B_1∨ ..∨ ¬B_n∨A_1 (n≥0)**
{
Build the temporary clause **C': ¬B_1∨...∨ ¬B_n**
Let G' to be the One-Successor of **G**
Compute R = C'∨ G'
Push R into the resolvent stack
}
}
if G is **empty ,** output **Yes else** output **No**

Figure 7

This method is similar to the standard SLD-Resolution method. The differences lie beyond data structure, the selection function and the important factor in OBDDs: the variable order.

We implemented and examined this algorithm for many logic programs. Unfortunately there are sev-

eral dependent factors that make analyzing the experimental results vary hard. The most important factors are: The logic program clause order, OBDD variable order, Nature of the logic program and the query, The logic program size, Number of propositional literals included in the logic program and A verage size of each logic program clause.

In order to analyze the experimental results we have to look for a suitable bench mark. At this moment we suffice to the intuitive reasonable results.

4.2.3 Resolution with Non-Linear Selection Functions

OBDDs are non-linear data structures, therefore we would expect to obtain better results if we try non-linear selection functions. In linear selection, when we have for example the goal $G: \neg A_1 \vee \ldots \vee \neg A_n$ we only consider the variable $\neg A_1$ and look for program clauses like: $C: \neg B_1 \vee \ldots \vee \neg B_n \vee A_1$ to resolve with.

Input: A logic program **P** and a goal **G** presented as a multi rooted **OBDD**

Output: Yes, if a proof of G from P was found, otherwise No.

Algorithm
Initialise the resolvent stack to be **G**, (the input goal)
while the resolvent stack is not empty
{
Pop a goal clause $G: \neg A_1 \vee \ldots \vee \neg A_n$
if G is empty **exit**
for all program clauses $C: \neg B_1 \vee \ldots \vee \neg B_n \vee A$
{ s.t. $A \in \{A_1 \ldots A_n\}$
Build C' from **C ;**
/* **C'** would be the same as **C** but with out **A** */
Build G' from **G ;**
/* **G'** would be the same as **G** but with $A_i = A$ */
Compute R = C' ∨ G'
If null condition has happened **exit**
Push R into the resolvent stack
}
}
if G is **empty** or **null condition has happened,** output **Yes** **else** output **No**

Figure 8

But we can also consider all the variables in $G: \neg A_1 \vee \ldots \vee \neg A_n$ and resolve **G** with any clause which includes one of the variables $A_1, \ldots, A_n$. In this case we should check for the **null condition**. This condition occurs only when we find a clause like $\neg P$ as a result of resolving two clauses and we had the unit clause P from the beginning. Figure 8, displays an abstract interpreter for this method.

This method sometimes works more efficient than the first method, but most times it leads to a larger OBDD. It seems that if this method will be applied with some useful heuristic then better or even interesting results will be obtained.

5 Conclusions

In this paper we introduced the application of BDDs in propositional logic programming. Also we introduced two methods of deduction for an OBDD based propositional logic program. The primary results are satisfiable, but we believe that in this topic there is much more potentials. In any case more investigation efforts seem to be valuable.

References

[1] C. Y. Lee: *Representation of switching circuits by binary decision programs*, The Bell Systems Technical Journal 38, 1959, pp. 985-999.

[2] S. B. Akers: *Binary Decision Diagrams,* IEEE Trans. On Computers 27,1978, pp. 509-516

[3] R. E. Bryant: *Graph Based Algorithms for Boolean Function Manipulation*, IEEE Trans. On Computers 35, 1986, pp. 677-695

[4] Ch. Meinel, T. Theobald: *Algorithms and Data Structures in VLSI Design, OBDD Fundamentals and Applications*, Springer, Heidelberg, 1998.

[5] E. Burke, E. Foxley: *Logic and its Applications,* Prentice Hall, 1996.

[6] R. L. Epstein, Predicate *logic,* Wadsworth, 2001.

[7] Ch. Meinel, H. Sack, V. Schillings, *VisBDD - A Web-based Visualization Framework for OBDD Algorithms*. IWLS 2002: 385-390

[8] The BDD Portal website : http://www.bdd-portal.org/